ULRR

Proving the correctness of unfold/fold program transformations using bisimulation

Item Type	Article
Authors	Hamilton, Geoff W.;Jones, Neil D.
Citation	Perspectives of Systems Informatics: Lecture Notes in Computer Science;7162 pp. 153-169
Publisher	Springer
Download date	2026-03-16 20:48:31
Item License	https://creativecommons.org/licenses/by-nc-sa/1.0/
Link to Item	https://hdl.handle.net/10344/2375

Proving the Correctness of Unfold/Fold Program Transformations Using Bisimulation

G.W. Hamilton

School of Computing
Dublin City University
Dublin 9, Ireland
e-mail: hamilton@computing.dcu.ie

Neil D. Jones

Computer Science Department
University of Copenhagen
2100 Copenhagen, Denmark
neil@diku.dk

Abstract. This paper shows that a bisimulation approach can be used to prove the correctness of unfold/fold program transformation algorithms. As an illustration, we show how our approach can be use to prove the correctness of positive supercompilation (due to Sørensen et al). Traditional program equivalence proofs show the original and transformed programs are contextually equivalent, i.e., have the same termination behaviour in all closed contexts. Contextual equivalence can, however, be difficult to establish directly.
Gordon and Howe use an alternative approach: to represent a program's behaviour by a labelled transition system whose bisimilarity relation is a congruence that coincides with contextual equivalence. Labelled transition systems are well-suited to represent global program behaviour.
On the other hand, unfold/fold program transformations use generalization and folding, and neither is easy to describe contextually, due to use of non-local information. We show that weak bisimulation on labelled transition systems gives an elegant framework to prove contextual equivalence of original and transformed programs. One reason is that folds can be seen in the context of corresponding unfolds.

1 Introduction

Unfold/fold program transformation techniques were first presented by Burstall and Darlington [3], and are used in many program transformation systems such as partial evaluation [9], deforestation [18] and supercompilation [17, 16]. Each of these program transformations apply (in some order) a sequence of meaning preserving rules, so the problem of proving that the transformations produce equivalent programs would appear to be trivial, but this is greatly complicated by the presence of folding. As a simple example: if function f is defined by $f = e$, then occurrences of the expression e can be replaced by calls to the function f in a folding step. However, if the occurrence of the expression e in this definition itself is folded, we obtain the non-terminating definition $f = f$. Thus unsupervised application of folding in any context may produce a program that is not equivalent to the original.

To avoid this problem we express transformation by semantics-preserving manipulation of labelled transition systems. Within our framework, folding is only

done with respect to proper ancestors in the transition system, thus avoiding the problem of folding $f = e$ into $f = f$. We eliminate intermediate data or function calls by removing silent transitions ($\xrightarrow{\tau}$) from the labelled transition system. We therefore use weak bisimulation for correctness proofs, based on a theorem that weak bisimulation is equivalent to contextual equivalence. This approach makes it easier to prove the correctness of unfold/fold transformations, as folds are seen in the context of corresponding unfolds. Further, correctness is decoupled from efficiency concerns, in contrast to Sands' theory of local improvement [14].

Plan: In Section 2 we define our higher-order functional language, and define a reduction semantics and contextual equivalence. In Section 3 we define labelled transition systems in general; a particular one for semantic analysis; and show that its weak bisimulation relation is equivalent to contextual equivalence. In Section 4 we use this framework to describe the positive supercompilation algorithm and show that it satisfies the correctness property. This requires an extended form of labelled transition system, one also equipped with fold transitions that rename program variables. In Section 5 we discuss related work and conclude. Appendices A, B and C define our own particular instances of homeomorphic embedding, expression generalization, and residualization which are used to define positive supercompilation within our framework.

2 Language

Definition 1 (Language Syntax). The simple higher-order functional language as shown in Fig. 1 is used throughout this paper.

$e ::= x$	Variable
$\mid c\ e_1 \ldots e_k$	Constructor Application
$\mid f$	Function Call
$\mid \lambda x.e$	λ-Abstraction
$\mid e_0\ e_1$	Application
$\mid \textbf{case}\ e_0\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$	Case Expression
$\mid e_0\ \textbf{where}\ f_1 = e_1 \ldots f_n = e_n$	Local Function Definition
$p ::= c\ x_1 \ldots x_k$	Pattern

Fig. 1. Language Grammar

A program in the language is an expression, which can be a variable, constructor application, function call, λ-abstraction, **case**, or **where**. Local functions are defined using **where**; it is assumed that these local definitions cannot contain any free variables. λ-abstracted variables and **case** expression pattern variables

are *bound*; all other variables are *free*. We use $fv(e)$ and $bv(e)$ to denote the free and bound variables respectively of expression e. We write $e_1 \equiv e_2$ if e_1 and e_2 differ only in the names of bound variables. We also write $e_1 \equiv e_2$ (MVR) if e_1 and e_2 are equivalent modulo variable renaming.

Each constructor has a fixed arity; for example *Nil* has arity 0 and *Cons* has arity 2. In an expression $c\ e_1 \ldots e_k$, k must equal the arity of c. Within the expression $\mathbf{case}\ e_0\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$, e_0 is called the *selector*, and $e_1 \ldots e_k$ are called the *branches*. The patterns in **case** expressions may not be nested. No variable may appear more than once within a pattern. We assume that the patterns in a **case** expression are non-overlapping and exhaustive.

Function environment Δ. Without loss of generality, we can assume that a program contains only one **where** clause, at the outermost level. For notational convenience we sometimes assume these have been collected into a function environment, denoted by $\Delta = \{f_1 = e_1, \ldots, f_n = e_n\}$.

Example 1. An example program to calculate the sum of the squares of a list of numbers xs is shown in Fig. 2. We employ the usual notation [] for *Nil* and $x : xs$ for *Cons* $x\ xs$. The functions *plus* and *square* in this program are assumed to be defined in an initial program environment.

$$
\begin{array}{l}
sum\ (squares\ xs) \\
\mathbf{where} \\
sum \quad\ = \lambda xs.sum'\ xs\ Zero \\
sum' \quad = \lambda xs.\lambda a.\,\mathbf{case}\ xs\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\quad [] \qquad\ \Rightarrow a \\
\qquad\qquad\qquad\qquad\ \mid x' : xs' \Rightarrow sum'\ xs'\ (plus\ a\ x') \\
squares = \lambda xs.\,\mathbf{case}\ xs\ \mathbf{of} \\
\qquad\qquad\qquad\quad [] \qquad\ \Rightarrow [] \\
\qquad\qquad\qquad\ \mid x' : xs' \Rightarrow (square\ x') : (squares\ xs')
\end{array}
$$

Fig. 2. Example Program: Sum of Squares

The operational semantics of the language is normal order reduction. Erroneous terms such as $(c\ e_1 \ldots e_k)\ e$ and $\mathbf{case}\ (\lambda x.e)\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$ are assumed not to occur.

Definition 2 (Substitution). We use the notation $\{x_1 := e_1, \ldots, x_n := e_n\}$ to denote a *substitution*. If e is an expression, then $e\{x_1 := e_1, \ldots, x_n := e_n\}$ is the result of simultaneously substituting the expressions $e_1, \ldots, e_n$ for the corresponding variables $x_1, \ldots, x_n$, respectively, in the expression e while ensuring that bound variables are renamed appropriately to avoid name capture.

Definition 3 (Context). A context C is an expression with a "hole" $[]$ in the place of one sub-expression (though not within a **where** clause). $C[e]$ is the expression obtained by replacing the hole in context C with the expression e. The free variables within e may become bound within $C[e]$; if $C[e]$ is closed then we call it a *closing context* for e.

The call-by-name operational semantics of our language is standard: define an evaluation relation $\Downarrow$ between closed expressions and *values*, where values are expressions in *weak head normal form* (i.e. constructor applications or λ-abstractions). We define a one-step reduction relation $\overset{r}{\leadsto}$ inductively as shown in Fig. 3, where the reduction r can be β (β-substitution), $=f$ (unfolding of function f) or κ (constructor elimination). We assume that the function definitions which are currently in scope are held in the environment Δ.

$$\frac{(f = e) \in \Delta}{f \overset{=f}{\leadsto} e} \qquad ((\lambda x.e_0)\ e_1) \overset{\beta}{\leadsto} (e_0\{x := e_1\}) \qquad \frac{e_0 \overset{r}{\leadsto} e_0'}{(e_0\ e_1) \overset{r}{\leadsto} (e_0'\ e_1)}$$

$$\frac{p_i = c\ x_1 \ldots x_n}{(\textbf{case}\ (c\ e_1 \ldots e_n)\ \textbf{of}\ p_1 : e_1' | \ldots | p_k : e_k') \overset{\kappa}{\leadsto} (e_i\{x_1 := e_1, \ldots, x_n := e_n\})}$$

$$\frac{e_0 \overset{r}{\leadsto} e_0'}{(\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 | \ldots p_k : e_k) \overset{r}{\leadsto} (\textbf{case}\ e_0'\ \textbf{of}\ p_1 : e_1 | \ldots p_k : e_k)}$$

Fig. 3. One-Step Reduction Relation

We use the notation $e \overset{r}{\leadsto}$ if the expression e reduces, $e \Uparrow$ if e diverges, $e \Downarrow$ if e converges and $e \Downarrow v$ if e evaluates to the value v. These are defined as follows, where $\overset{r}{\leadsto}^*$ denotes the reflexive transitive closure of $\overset{r}{\leadsto}$:

$$e \overset{r}{\leadsto}, \text{ iff } \exists e'.e \overset{r}{\leadsto} e' \qquad e \Downarrow, \text{ iff } \exists v.e \Downarrow v$$
$$e \Downarrow v, \text{ iff } e \overset{r}{\leadsto}^* v \wedge \neg(v \overset{r}{\leadsto}) \qquad e \Uparrow, \text{ iff } \forall e'.e \overset{r}{\leadsto}^* e' \Rightarrow e' \overset{r}{\leadsto}$$

Definition 4 (Contextual Equivalence). Contextual equivalence, denoted by $\simeq$, equates two expressions if and only if they exhibit the same termination behaviour in all closing contexts i.e. $e_1 \simeq e_2$ iff $\forall C\ .\ C[e_1] \Downarrow$ iff $C[e_2] \Downarrow$.

3 Bisimulation

We first give standard definitions of *labelled transition system* and *weak bisimulation*, and then show how they can be used to describe runtime states of a functional program.

Definition 5 (Labelled Transition System). A labelled transition system (LTS for short) is a tuple $\Sigma = (\mathcal{S}, s_{init}, \delta, Act)$ where:

- $\mathcal{S}$ is the set of *states*.
- $s_{init} \in \mathcal{S}$ is the start state.
- the set of *actions* $\alpha \in Act$ include the *silent transition* τ.
- *transition relation* $\delta \subseteq \mathcal{S} \times Act \times \mathcal{S}$ relates pairs of states by actions.
- Notation: as usual, write $s \xrightarrow{\alpha} s'$ in place of $(s, \alpha, s') \in \delta$. We denote the set of *all* non-silent transitions from state s by $s \to (\alpha_1, s_1) \ldots (\alpha_n, s_n)$ (a simpler notation than the multilevel $\{s \xrightarrow{\alpha_1} s_1, \ldots, s \xrightarrow{\alpha_n} s_n\}$ or highly-parenthesized $\{(s, \alpha_1, s_1), \ldots, (s, \alpha_n, s_n))\}$.)

We write $s \Rightarrow s'$ iff there is a (possibly empty) sequence of silent transitions leading from s to s'. For each action α, we write $s_1 \stackrel{\alpha}{\Rightarrow} s_2$ iff there are s_1' and s_2' such that $s_1 \Rightarrow s_1' \xrightarrow{\alpha} s_2' \Rightarrow s_2$.

Definition 6 (Weak Simulation). A binary relation $\mathcal{R} \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ is a *weak simulation* of labelled transition system $\Sigma_1 = (\mathcal{S}_1, s_{init_1}, \delta_1, Act_1)$ by $\Sigma_2 = (\mathcal{S}_2, s_{init_2}, \delta_2, Act_2)$ if $(s_{init_1}, s_{init_2}) \in \mathcal{R}$, and for every pair $(s_1, s_2) \in \mathcal{R}$:

$$\forall \alpha \in Act_1, s_1' \in \mathcal{S}_1.\ \text{if } (s_1 \stackrel{\alpha}{\Rightarrow} s_1') \in \delta_1 \text{ then } \exists s_2' \in \mathcal{S}_2\ .\ (s_2 \stackrel{\alpha}{\Rightarrow} s_2') \in \delta_2 \wedge (s_1', s_2') \in \mathcal{R}$$

Definition 7 (Weak Bisimulation). A *weak bisimulation* is a binary relation $\mathcal{R}$, where both $\mathcal{R}$ and its inverse $\mathcal{R}^{-1}$ are weak simulations.

Definition 8 (Weak Bisimilarity). If there exists any weak bisimulation $\mathcal{R}$ between labelled transition systems Σ_1 and Σ_2, then there exists a unique maximal one, henceforth denoted by $\sim$. A notation: we also write $s_1 \sim s_2$ in place of $(s_1, s_2) \in \sim$.

In the spirit of Gordon [5] we now define a particular labelled transition system that characterises the immediate observations that can be made on expressions to determine their observational equivalence. We extend [5] by allowing free variables in expressions and thus also in actions. Observational equivalence will therefore require that the free variables in actions match up (bound variables must also match up, but this can be done by renaming).

Definition 9 (Driven LTS). Fig. 4 defines $\mathcal{D}[\![e]\!] = (\mathcal{S}, e_0, \to, Act_e)$ to be the *driven LTS* associated with program e_0. Here,

- Act_e is a set of actions with possible forms: v, c, @, λv, $\#i$, **case**, p and τ.
  An action may be: x, a variable; c, a constructor; @, a function application; $\#i$, the ith argument in an application; λx, an abstraction over variable x; **case**, a case selector; or p, a case branch pattern. Bound variables may have been renamed to avoid name clashes.
- $\mathcal{S} \subseteq (Exp \cup \{\mathbf{0}\})$ and $\to \subseteq Exp \times Act_e \times (Exp \cup \{\mathbf{0}\})$ are the smallest sets such that $e_0 \in \mathcal{S}$ and $\to$ satisfies Fig. 4. (Note that $e \xrightarrow{\alpha} e'$ means $(e, \alpha, e') \in \to$. When convenient we use the compact transition notation of Definition 5.)

1. Root and branch growth: $e_0 \in \mathcal{S}$. If $e \in \mathcal{S}$ and $e \xrightarrow{\alpha} e'$ then $e' \in \mathcal{S}$.

2. Functions: If $f \in \mathcal{S}$ and $(f = e) \in \Delta$ then $f \xrightarrow{\tau} e$

3. Applications:
   (a) If $e = x\ e_1 \dots e_n \in \mathcal{S}$ then $e \rightarrow (x, \mathbf{0})(\#1, e_1) \dots (\#n, e_n)$
   (b) If $e = c\ e_1 \dots e_n \in \mathcal{S}$ then $e \rightarrow (c, \mathbf{0})(\#1, e_1) \dots (\#n, e_n)$
   (c) If $e = (\lambda x.e_0)\ e_1 \in \mathcal{S}$ then $e \xrightarrow{\tau} e_0\{x := e_1\}$
   (d) If $e = e_0\ e_1 \in \mathcal{S}$ and $e_0 \rightarrow (\mathbf{case}, e'_0)(p'_1, e'_1) \dots (p'_n, e'_n)$ then $e \rightarrow (\mathbf{case}, e'_0)(p'_1, e'_1\ e_1) \dots (p'_n, e'_n\ e_1)$
   (e) Otherwise, if $e = e_0\ e_1 \in \mathcal{S}$ and $e_0 \xrightarrow{\tau} e'_0$ then $e_0\ e_1 \xrightarrow{\tau} e'_0\ e_1$

4. **case**: If $\mathcal{S} \ni e = (\mathbf{case}\ e_0\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k)$ then
   (a) If $e_0 = x\ e_1 \dots e_n$ and $e'_i = e''_i\{x' := e_0\}$ then $$e \rightarrow (\mathbf{case}, e_0)(p_1, e''_1\{x' := p_1\}) \dots (p_k, e''_k\{x' := p_k\})$$
   (b) If $e_0 = c\ e_1 \dots e_n$ and $p_i = c\ x_1 \dots x_n$ then $e \xrightarrow{\tau} e'_i\{x_1 := e_1, \dots, x_n := e_n\}$
   (c) If $e_0 \rightarrow (\mathbf{case}, e'_0)(p'_1, e'_1) \dots (p'_n, e'_n)$ then $$e \rightarrow (\mathbf{case}, e'_0)(p'_1, \mathbf{case}\ e'_1\ \mathbf{of}\ p_1 \Rightarrow e_1| \dots |p_k \Rightarrow e_k) \dots (p'_n, \mathbf{case}\ e'_n\ \mathbf{of}\ p_1 \Rightarrow e_1| \dots |p_k \Rightarrow e_k)$$
   (d) Otherwise, if $e_0 \xrightarrow{\tau} e'_0$ then $e \xrightarrow{\tau} (\mathbf{case}\ e'_0\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k)$

5. λ-abstraction: If $\lambda x.e \in \mathcal{S}$ then $\lambda x.e \xrightarrow{\lambda x} e$

Fig. 4. The Driven Labelled Transition System $\mathcal{D}[\![e]\!]$

$\mathbf{0}$ corresponds to a state from which there are no transitions; transitions labelled with a variable or a constructor will lead into this state.
Note on rules 2, 3c, 4b: function unfolding, β-reduction and constructor elimination are not relevant to the observational equivalence of original and transformed programs. Thus they are represented by the silent transition τ, and weak bisimulation is appropriate for comparing program behaviour. All function applications are removed by driving, so no @ actions will appear in the driven LTS; these actions are only introduced as a result of generalization, described later.

Example 2. A portion of the driven LTS $\mathcal{D}[\![e]\!]$ constructed for the program in Fig. 2 is shown in Fig. 5. Functions *plus* and *square* are treated as free variables.

A central property: the weak bisimilarity relation $\sim$ between $\mathcal{D}[\![e]\!]$ and $\mathcal{D}[\![e']\!]$ is a congruence, and coincides with contextual equivalence.

Theorem 1 (Congruence). $\forall C\ .\ e \sim e' \Rightarrow C[e] \sim C[e']$

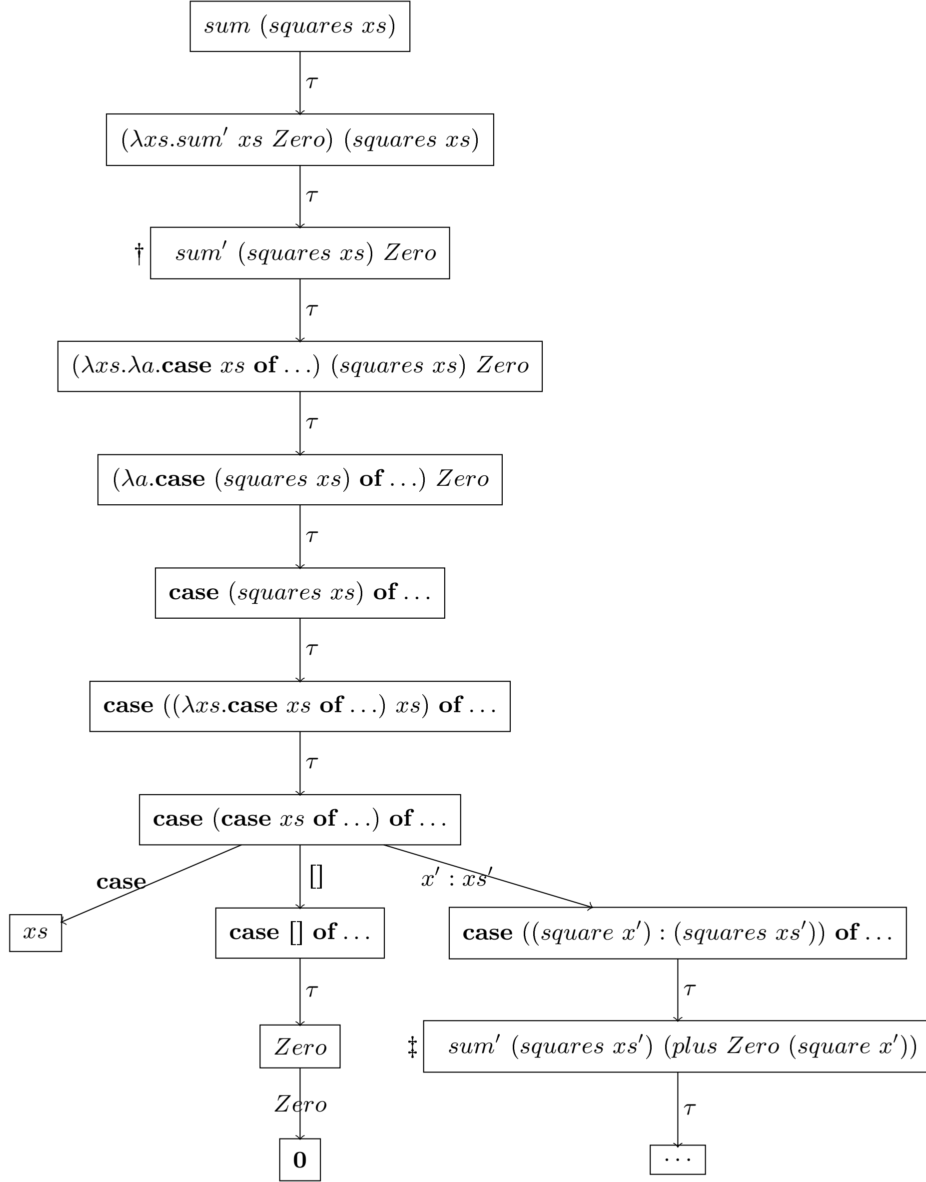


Fig. 5. Portion of the infinite LTS $\mathcal{D}[\![e]\!]$ resulting from Driving *sum* (*squares xs*)

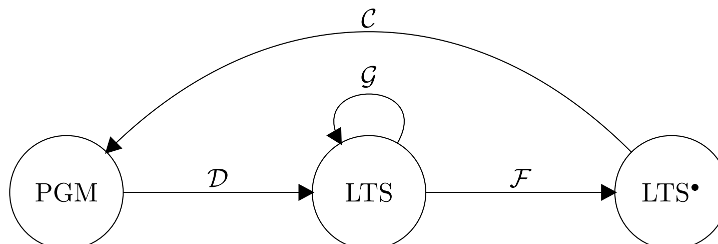
Proof. Similar to that of Howe [8]; not given here due to space constraints.

Theorem 2 (Operational Extensionality). $\simeq \, = \, \sim$

Proof. The proof that $\sim \, \subseteq \, \simeq$ follows from the congruence of $\sim$. The reverse inclusion follows by co-induction after showing that $\simeq$ is a bisimulation on $\mathcal{D}[\![e]\!]$.

4 Positive Supercompilation

The positive supercompilation algorithm computes $\mathcal{T}[\![e]\!] = \mathcal{F}[\![\mathcal{G}[\![\mathcal{D}[\![e]\!]]\!]]\!]$. We show correctness of the more powerful distillation algorithm [6] in a future paper.



Given a program $e \in$ PGM, the driving rules $\mathcal{D}$ of **Fig.** 4 are applied, beginning with the root e, to construct the labelled transition system $\mathcal{D}[\![e]\!]$. Although $\mathcal{D}[\![e]\!]$ can be infinite in general, our unfold/fold program transformer will traverse only finite portions by working lazily from the root. If a *danger of infinite unfolding* is detected ("the whistle is blown"), then generalization rules $\mathcal{G}$ are applied to transform the current labelled transition system into a new version without local danger of infinite unfolding. Overall, the effect is to transform $\mathcal{D}[\![e]\!]$ into an LTS $\mathcal{G}[\![\mathcal{D}[\![e]\!]]\!]$ that has only *a finite number of different expressions on any path* from its root (MVR, i.e., modulo variable renaming).

Finally, folding rules $\mathcal{F}$ are applied to this generalized LTS to produce a *folded labelled transition system* (a so-called LTS$^\bullet$) that contains *only a finite number of states.* A residual program can be constructed from this finite folded labelled transition system using the $\mathcal{C}$ rules defined in Section C. Its syntax and efficiency may be substantially different from those of the original program.

4.1 Generalization

A suitable *homeomorphic embedding* relation $\lesssim$ is defined in Appendix A, analogous to Sørensen's [16] but adapted to our language. Its essential property:

Lemma 1. *There exists a computable partial order $\lesssim$ on Exp such that in any infinite sequence of expressions $e_0, e_1, \ldots$ there exists some $i < j$ where $e_i \lesssim e_j$.*

By Lemma 1, if the set of paths from the root of $\mathcal{D}[\![e]\!]$ is infinite, an instance of homeomorphic embedding will occur, and can be computably detected while constructing $\mathcal{D}[\![e]\!]$. Generalization is performed while traversing $\mathcal{D}[\![e]\!]$ from its root. If "the whistle blows," a state that is a homeomorphic embedding of one of its ancestors will be generalized.

The first transformation step is to build from $\mathcal{D}[\![e]\!]$ a computationally equivalent new LTS with generalizations added by the insertion of @ transitions, representing the application of a generalized expression to the sub-terms which have been extracted from it. This generalization is sufficient to ensure that the

only homeomorphic embeddings that remain are also renamings. Generalization by the insertion of @ transitions is performed by the *abstract* operation, defined in Appendix B. Adequacy is expressed by the following result:

Lemma 2 (Abstraction Lemma). *If $e' \lesssim e$ and $e' \not\equiv e$ (MVR), then there exists an expression $e'' = abstract(e, e') = (\lambda x_1 \ldots x_n.e_0)\ e_1 \ldots e_n$ such that $e'' \overset{\beta^*}{\leadsto} e$ and, for $i = 0, 1, \ldots, n$, $e \not\lesssim e_i$.*

This is equivalent to replacing e by **let** $x_1 = e_1, \ldots, x_n = e_n$ **in** e_0 (an idea from Turchin [17]). The node e in the LTS is therefore replaced by the following:

$$e'' \to (@, \lambda x_1 \ldots x_n.e_0)(\#1, e_1) \ldots (\#n, e_n)$$

Such generalized expressions are not further reduced by driving; driving is applied only to their sub-expressions e_i. Replacements (such as e by e'') are repeated until the LTS converges to a version in which the only homeomorphic embeddings along any path from the root of $\mathcal{D}[\![e']\!]$ are also renamings. This process terminates since each e_i is smaller than e. By König's Lemma, once it does terminate, the resulting LTS will have a finite set of nodes.

We will prove Lemma 1 and Lemma 2 in Appendices A and B, so these transformations are correct within the bisimulation framework.

Example 3. We generalize the labelled transition system in Fig. 5. Consider

$$e_\dagger = sum'\ (squares\ xs)\ Zero$$

(the expression at node †). Now $e_\dagger$ is homeomorphically embedded in expression

$$e_\ddagger = sum'\ (squares\ xs')\ (plus\ Zero\ (square\ x'))$$

at node ‡, i.e., $e_\dagger \lesssim e_\ddagger$. Thus we generalize $e_\ddagger$ with respect to $e_\dagger$ to give:

$$e'' = (\lambda v.sum'\ (squares\ xs')\ v)\ (plus\ Zero\ (square\ x'))$$

Edges $e'' \to (@, \lambda v.sum'\ (squares\ xs')\ v)(\#1, plus\ Zero\ (square\ x'))$ are added; and these subexpression are then further driven and generalized. The portion of the resulting LTS rooted at e'' is shown in Fig. 6.

Definition 10 (Generalization Algorithm). *Here e, s range over expressions.*

$\mathcal{G}[\![e]\!] = \mathcal{G}'[\![e]\!]\{\}$

If $e \xrightarrow{\tau} e'$ then

$$\mathcal{G}'[\![e]\!]\ \rho = \begin{cases} e \xrightarrow{\tau} e', & \text{if } \exists e'' \in \rho\ .\ e'' \equiv e \text{ (MVR)} \\ (e_0\ e_1 \ldots e_n) \to (@, e_0')(\#1, e_1') \ldots (\#n, e_n'), & \text{if } \exists e'' \in \rho\ .\ e'' \lesssim e \\ \text{where } e_0\ e_1 \ldots e_n = abstract(e, e'') & \\ \qquad \forall i \in \{0 \ldots n\}\ .\ e_i' = \mathcal{G}'[\![\mathcal{D}[\![e_i]\!]]\!]\ \rho & \\ e \xrightarrow{\tau} \mathcal{G}'[\![e']\!]\ (\rho \cup \{e\}), & \text{otherwise} \end{cases}$$

If $e \to (\alpha_1, e_1) \ldots (\alpha_n, e_n)$ then
$\mathcal{G}'[\![e]\!]\ \rho = e \to (\alpha_1, \mathcal{G}'[\![e_1]\!]\ \rho) \ldots (\alpha_n, \mathcal{G}'[\![e_n]\!]\ \rho)$

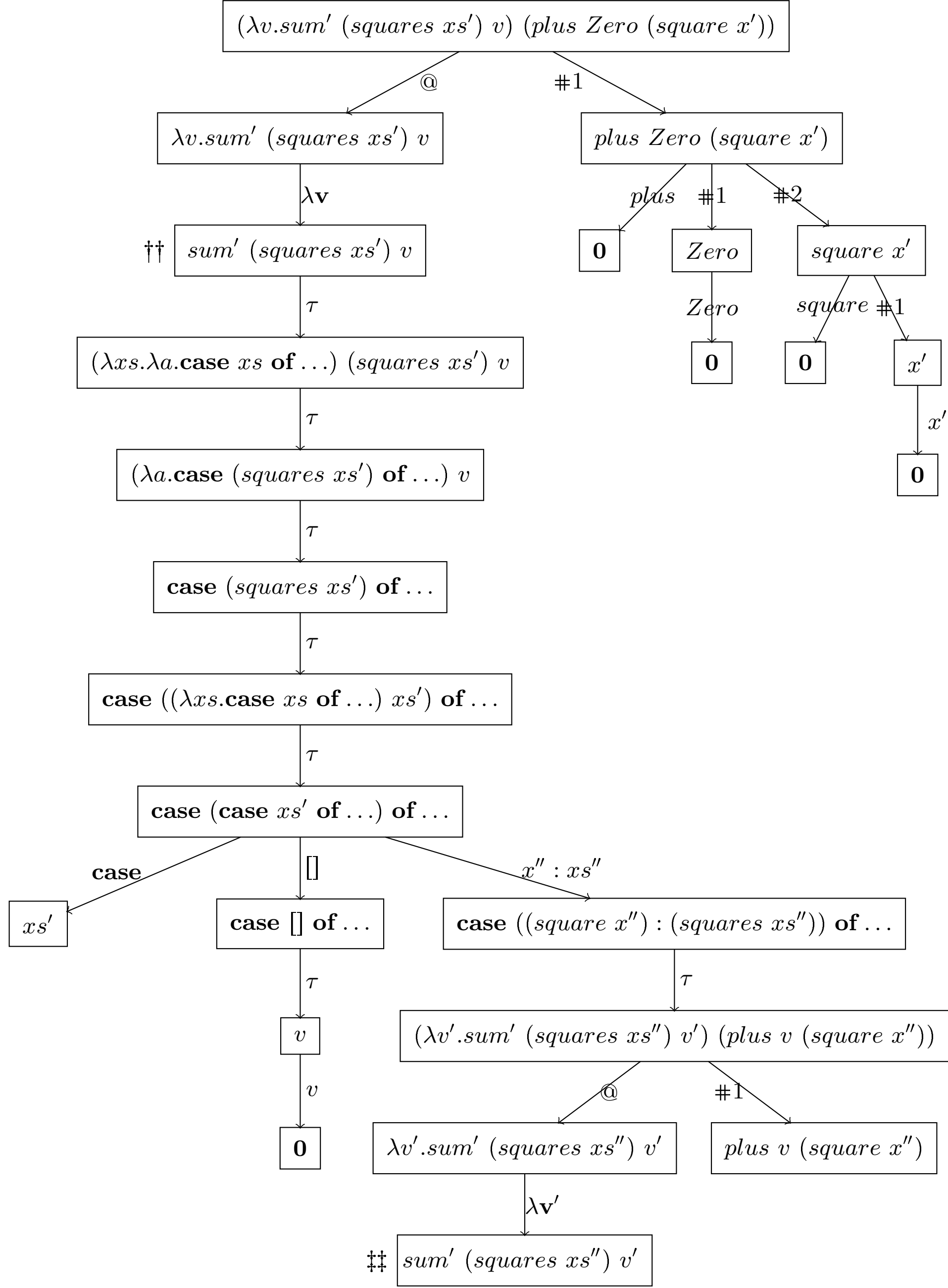


Fig. 6. Portion of LTS $\mathcal{G}[\![\mathcal{D}[\![e]\!]]\!]$ resulting from Generalizing $\mathcal{D}[\![e]\!]$

Style explanation in **Definition** 10:
– $\mathcal{G}, \mathcal{G}'$ each transform a rooted input LTS into a rooted output LTS.

– The algorithm constructs the transitions in the output LTS.
– Starting at the root e of the input LTS, $\mathcal{G}'$ inspects the transitions from e.
– When $\mathcal{G}'$ calls itself recursively, the output LTS includes the union of the output LTS's resulting from the recursive calls.

Content explanation in **Definition** 10:
– Transformation of the labelled transition system $\mathcal{D}[\![e]\!]$ begins at the root e.
– The set ρ is always a "history": the set of expressions from which silent transitions have been taken since traversing the root.
– To start, $\mathcal{G}[\![e]\!]$ calls $\mathcal{G}'[\![e]\!]$ with an empty history.
– For any expression e from which a silent transition issues:
  • $\mathcal{G}$ terminates if the source expression has already been seen (MVR)
  • if some ancestor e' is embedded in e then generalization is done; and the resulting expression components are driven and generalized further
– otherwise, $\mathcal{G}$ recursively processes all the children of expression e
– It suffices to do the embedding check only at silent transitions, since any infinite transition sequence would contain infinitely many silent transitions.

4.2 Folding

Let $\Sigma = \mathcal{G}[\![\mathcal{D}[\![e]\!]]\!]$ be the labelled transition system resulting from driving and generalization. The next step is to construct a bisimilar LTS $\Sigma^\bullet$ that has only a finite number of states in all. (Relatively easy to do, after generalization has done the hard work!) We adapt the classical folding technique to labelled transition systems. First, extend the definition of LTS by allowing *fold transitions* of the form $e \stackrel{\theta}{\to} e'$. Here e is an expression, e' is one of its ancestors in Σ, and θ is a renaming such that $e \equiv e'\ \theta$.

Definition 11 (Folded LTS). *A folded LTS has form* $\Sigma^\bullet = (\mathcal{S}, e_{init}, \to^\bullet, Act^\bullet)$ *where*

$$Act^\bullet = Act_e \cup \{\ \theta \mid \theta \textit{ is a renaming}\ \}$$

A folded transition $e \stackrel{\theta}{\to} e'$ will generate a call to a residual function in the program output that supercompilation produces, see Appendix C.

4.3 The Folding Transformation

Definition 12 (Folding Algorithm). *This takes Σ into $\Sigma^\bullet$. The structure is similar to $\mathcal{G}$ (but simpler). We write $\to$ for both input and output transitions.*

$$\mathcal{F}[\![e]\!] = \mathcal{F}'[\![e]\!]\{\}$$

If $e \stackrel{\tau}{\to} e'$ then

$$\mathcal{F}'[\![e]\!]\ \rho = \begin{cases} e \stackrel{\theta}{\to} e'', & \text{if } \exists e'' \in \rho\ .\ e'' \equiv e \text{ (MVR)} \wedge e'' \equiv e\theta \\ e \stackrel{\tau}{\to} \mathcal{F}'[\![e']\!]\ (\rho \cup \{e\}), & \text{otherwise} \end{cases}$$

If $e \to (\alpha_1, e_1) \ldots (\alpha_n, e_n)$ then

$$\mathcal{F}'[\![e]\!]\ \rho = e \to (\alpha_1, \mathcal{F}'[\![e_1]\!]\ \rho) \ldots (\alpha_n, \mathcal{F}'[\![e_n]\!]\ \rho)$$

Within these rules, ρ is the set of previously encountered expressions. If an expression is encountered that is a renaming of one of them, then folding is performed. Since any infinite transition sequence has infinitely many silent transitions, the renaming check is only done at silent transitions.

The result of the above steps will be a labelled transition system $\Sigma^\bullet$ with folding. We need to show that the LTS for the original program is equivalent to the labelled transformation system with folding resulting from its transformation. First, a couple of definitions are needed.

Definition 13 (Folded Weak Simulation). Binary relation $\mathcal{R} \subseteq Exp \times Exp$ is a *folded weak simulation* of folded LTS $\Sigma_1 = (Exp, e_{init_1}, \delta_1, Act^\bullet)$ by folded LTS $\Sigma_2 = (Exp, e_{init_2}, \delta_2, Act^\bullet)$ if $(e_{init_1}, e_{init_2}) \in \mathcal{R}$, and for every $(e_1, e_2) \in \mathcal{R}$:
(a) $\forall e_1' \in Exp, \alpha \in Act_e$. if $(e_1 \xrightarrow{\alpha} e_1') \in \delta_1$ then $\exists (e_2 \stackrel{\alpha}{\Rightarrow} e_2') \in \delta_2$. $(e_1', e_2') \in \mathcal{R}$
(b) $\forall e_1' \in Exp, \theta$. if $(e_1 \xrightarrow{\theta} e_1') \in \delta_1$ then $(e_1', e_2\theta) \in \mathcal{R}$

Definition 14 (Folded Weak Bisimulation). A *folded weak bisimulation* is a binary relation $\mathcal{R}$, where both $\mathcal{R}$ and its inverse $\mathcal{R}^{-1}$ are folded weak simulations.

Theorem 3 (Correctness of Folding). *There is a folded weak bisimulation $\mathcal{R}$ between Σ and $\mathcal{F}[\![\Sigma]\!]$.*

Proof. Proof is by induction on length of paths from the roots of Σ and $\mathcal{F}[\![\Sigma]\!]$.

Example 4. Applying the folding pass to the labelled transition system in Fig. 6, the expression at node ‡‡ is a renaming of the expression at node ††. The expression at node ‡‡ $(sum' \ (squares \ xs'') \ v')$ is therefore folded into node ††, with the renaming $\{xs' := xs'', v := v'\}$.

The program of Fig. 7 is constructed from this labelled transition system with folding, using the rules shown in Appendix C.

$$
\begin{array}{l}
\textbf{case}\ xs\ \textbf{of} \\
\quad [] \qquad \Rightarrow Zero \\
\quad |\ x' : xs' \Rightarrow f\ xs'\ (plus\ Zero\ (square\ x')) \\
\textbf{where} \\
f = \lambda xs'.\lambda v.\,\textbf{case}\ xs'\ \textbf{of} \\
\qquad\qquad\qquad [] \qquad \Rightarrow v \\
\qquad\qquad\qquad |\ x'' : xs'' \Rightarrow f\ xs''\ (plus\ v\ (square\ x''))
\end{array}
$$

Fig. 7. Supercompiled Example Program: Sum of Squares

5 Conclusion and Related Work

In this paper, we have described a new approach to proving the correctness of unfold/fold transformations. We have defined a labelled transition system semantics for programs to represent their behaviour and a weak bisimulation relation between these LTSs to show their observational equivalence. We then proved that this weak bisimulation implies contextual equivalence. We argue that this approach makes it easier to prove the correctness of unfold/fold transformations as folds can be seen in the context of corresponding unfolds, and correctness is also decoupled from efficiency concerns.

The seminal work in the area of proving the correctness of unfold/fold program transformations is Sands' theory of *local improvement* [14]. Using this approach, the correctness of program transformations is linked to showing the improvement in efficiency of local transformations. However, this is complicated by the use of folding, which causes a loss of efficiency locally, but not globally if it is always done in conjunction with a corresponding unfold. Also, tying the correctness of transformations to an improvement in efficiency restricts this approach to particular program semantics and transformation techniques.

Bisimilarity has been applied to functional programming languages before, notably by Abramsky in his study of applicative bisimulation and the lazy λ-calculus [1], and by Howe who developed a powerful method for showing that bisimilarity is a congruence [8]. Both showed that their definitions of bisimilarity are equal to contextual equivalence (operational extensionality). A labelled transition system semantics was first defined directly for a functional language by Gordon [5]. This simplified the definition of bisimulation and allowed a lot of the techniques used for process algebras to be applied to functional languages. We have extended Gordon's LTS semantics to incorporate additional language constructs and also to allow terms that contain free variables. Bisimulation has previously been used to prove the correctness of program transformations for imperative languages, which lend themselves more naturally to a labelled transition system semantics [11]. The focus of all the previous work on applying bisimulation techniques to functional programs was not on proving the correctness of program transformations. In this paper, we show how a slight modification to the definition of bisimulation allows it to be applied to the results of unfold/fold program transformations, thus giving us a straightforward technique for proving their correctness.

Acknowledgements

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre (www.lero.ie), and by the School of Computing, Dublin City University.

References

1. Abramsky, S.: The lazy lambda calculus. In: Research Topics in Functional Programming. pp. 65–116. Addison-Wesley (1990)
2. Bol, R.: Loop Checking in Partial Deduction. Journal of Logic Programming 16(1–2), 25–46 (1993)
3. Burstall, R., Darlington, J.: A transformation system for developing recursive programs. Journal of the ACM 24(1), 44–67 (Jan 1977)
4. Dershowitz, N., Jouannaud, J.P.: Rewrite Systems. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, pp. 243–320. Elsevier, MIT Press (1990)
5. Gordon, A.D.: Bisimilarity as a theory of functional programming. Theoretical Computer Science 228(1–2), 5–47 (1999)
6. Hamilton, G.: Distillation: Extracting the Essence of Programs. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation. pp. 61–70 (2007)
7. Higman, G.: Ordering by Divisibility in Abstract Algebras. Proceedings of the London Mathematical Society 2, 326–336 (1952)
8. Howe, D.J.: Proving congruence of bisimulation in functional programming languages. Information and Computation 124(2), 103–112 (February 1996)
9. Jones, N., Gomard, C., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall (1993)
10. Kruskal, J.: Well-Quasi Ordering, the Tree Theorem, and Vazsonyi's Conjecture. Transactions of the American Mathematical Society 95, 210–225 (1960)
11. Lacey, D., Jones, N.D., Wyk, E.V., Frederiksen, C.C.: Compiler optimization correctness by temporal logic. Higher-Order and Symbolic Computation 17(3), 173–206 (2004)
12. Leuschel, M.: On the Power of Homeomorphic Embedding for Online Termination. In: Proceedings of the International Static Analysis Symposium, Pisa, Italy. pp. 230–245 (1998)
13. Marlet, R.: Vers une Formalisation de l'Évaluation Partielle. Ph.D. thesis, Université de Nice - Sophia Antipolis (1994)
14. Sands, D.: Proving the Correctness of Recursion-Based Automatic Program Transformations. Theoretical Computer Science 167(1–2), 193–233 (1996)
15. Sørensen, M.H., Glück, R.: An Algorithm of Generalization in Positive Supercompilation. Lecture Notes in Computer Science 787, 335–351 (1994)
16. Sørensen, M.H., Glück, R., Jones, N.: A Positive Supercompiler. Journal of Functional Programming 6(6), 811–838 (1996)
17. Turchin, V.: Program Transformation by Supercompilation. Lecture Notes in Computer Science 217, 257–281 (1985)
18. Wadler, P.: Deforestation: Transforming programs to eliminate trees. Lecture Notes in Computer Science 300, 344–358 (1988)

A Homeomorphic Embedding for Supercompilation

Generalization is performed when an expression is encountered that is an *embedding* of a previously encountered one. This is done by *homeomorphic embedding*, which we denote using $\lesssim$. The homeomorphic embedding relation was derived from results by Higman [7] and Kruskal [10] and was defined within term rewriting systems [4] for detecting the possible divergence of the term rewriting process.

Variants of this relation have been used to ensure termination within positive supercompilation [15], partial evaluation [13] and partial deduction [2, 12].

Definition 15 (Well-Quasi Order). A well-quasi order on a set S is a reflexive, transitive relation $\lesssim$ such that for any infinite sequence $s_1, s_2, \ldots$ of elements from S there are numbers i, j with $i < j$ and $s_i \lesssim s_j$.

This ensures that in any infinite sequence of expressions $e_0, e_1, \ldots$ there definitely exists some $i < j$ where $e_i \lesssim e_j$, so an embedding must eventually be encountered and transformation will not continue indefinitely.

Definition 16 (Embedding of Expressions). To define our homeomorphic embedding relation on expressions $\lesssim$, we first define a relation $\trianglelefteq$, where $e_1 \trianglelefteq e_2$ if all of the free variables within e_1 and e_2 match up and $FV = fv(e_1)$.

$$\frac{e_1 \bowtie e_2}{e_1 \trianglelefteq e_2} \qquad \frac{e_1 \lhd e_2 \quad fv(e_1) \subseteq FV}{e_1 \trianglelefteq e_2}$$

$$v \bowtie v \qquad f \bowtie f$$

$$\frac{\forall i \in \{1 \ldots n\}.e_i \trianglelefteq e'_i}{(c\ e_1 \ldots e_n) \bowtie (c\ e'_1 \ldots e'_n)} \qquad \frac{\exists i \in \{1 \ldots n\}.e \trianglelefteq e_i}{e \lhd (c\ e_1 \ldots e_n)}$$

$$\frac{e \trianglelefteq (e'\{x' := x\})}{\lambda x.e \bowtie \lambda x'.e'} \qquad \frac{e \trianglelefteq e'}{e \lhd \lambda x.e'}$$

$$\frac{e_0 \bowtie e'_0 \quad e_1 \trianglelefteq e'_1}{(e_0\ e_1) \bowtie (e'_0\ e'_1)} \qquad \frac{\exists i \in \{0, 1\}.e \trianglelefteq e_i}{e \lhd (e_0\ e'_1)}$$

$$\frac{e_0 \trianglelefteq e'_0 \quad \forall i \in \{1 \ldots n\}.\exists \theta_i.p_i \equiv (p'_i\ \theta_i) \wedge e_i \trianglelefteq (e'_i\ \theta_i)}{(\mathbf{case}\ e_0\ \mathbf{of}\ p_1 : e_1 | \ldots | p_n : e_n) \bowtie (\mathbf{case}\ e'_0\ \mathbf{of}\ p'_1 : e'_1 | \ldots | p'_n : e'_n)}$$

$$\frac{\exists i \in \{0 \ldots n\}.e \trianglelefteq e_i}{e \lhd (\mathbf{case}\ e_0\ \mathbf{of}\ p_1 : e_1 | \ldots | p_n : e_n)}$$

An expression is embedded within another by this relation if either *diving* (denoted by $\lhd$) or *coupling* (denoted by $\bowtie$) can be performed. Diving occurs when an expression is embedded in a sub-expression of another expression, and coupling occurs when two expressions have the same top-level construct and all the corresponding sub-expressions of the two constructs are embedded. Our version of this embedding relation extends previous versions to handle λ-abstractions and **case** expressions that contain bound variables. In these instances, the bound variables within the two expressions must also match up. Diving cannot be applied if the embedded expression contains any bound variables without their corresponding binders; this avoids the possibility of extracting these variables outside of their binders. The homeomorphic embedding relation $\lesssim$ can now be defined as follows:

$$e_1 \lesssim e_2 \text{ iff } \exists e.e_1 \equiv e(MVR) \wedge e \bowtie e_2$$

A technical point: within this relation, the two expressions must be coupled, but there is no longer a requirement that all of the free variables within the two expressions match up. Generalizing only when two expressions are coupled ensures that the result is not a variable, and there is no need for a *split* operation as used in [15]. It can be shown that the homeomorphic embedding relation is a well-quasi-order.

Lemma 1.
Proof. This is very similar to the proof of [4].

B Generalization Algorithm for Supercompilation

Definition 17 (Generalization of Expressions). The *generalization* of two expressions e_1 and e_2 is a triple $(e_g, \theta_1, \theta_2)$ where θ_1 and θ_2 are substitutions such that $e_g\theta_1 \equiv e_1$ and $e_g\theta_2 \equiv e_2$. This generalization is defined as follows:

$$
\begin{array}{l}
(x\ e_1 \ldots e_n) \sqcap (x\ e'_1 \ldots e'_n) = (x\ e^g_1 \ldots e^g_n, \bigcup_{i=1}^n \theta_i, \bigcup_{i=1}^n \theta'_i) \\
\quad \text{where} \\
\quad \forall i \in \{1 \ldots n\}.(e^g_i, \theta_i, \theta'_i) = e_i \sqcap e'_i \\
(c\ e_1 \ldots e_n) \sqcap (c\ e'_1 \ldots e'_n) = (c\ e^g_1 \ldots e^g_n, \bigcup_{i=1}^n \theta_i, \bigcup_{i=1}^n \theta'_i) \\
\quad \text{where} \\
\quad \forall i \in \{1 \ldots n\}.(e^g_i, \theta_i, \theta'_i) = e_i \sqcap e'_i \\
(f\ e_1 \ldots e_n) \sqcap (f\ e'_1 \ldots e'_n) = (f\ e^g_1 \ldots e^g_n, \bigcup_{i=1}^n \theta_i, \bigcup_{i=1}^n \theta'_i) \\
\quad \text{where} \\
\quad \forall i \in \{1 \ldots n\}.(e^g_i, \theta_i, \theta'_i) = e_i \sqcap e'_i \\
((\lambda x.e_0)\ e_1 \ldots e_n) \sqcap ((\lambda x'.e'_0)\ e'_1 \ldots e'_n) = ((\lambda x.e^g_0)\ e^g_1 \ldots e^g_n, \bigcup_{i=0}^n \theta_i, \bigcup_{i=0}^n \theta'_i) \\
\quad \text{where} \\
\quad (e^g_0, \theta_0, \theta'_0) = e_0 \sqcap (e'_0\{x' := x\}) \\
\quad \forall i \in \{1 \ldots n\}.(e^g_i, \theta_i, \theta'_i) = e_i \sqcap e'_i \\
((\mathbf{case}\ e_0\ \mathbf{of}\ p_1 : e_1 | \ldots | p_k : e_k)\ e_{k+1} \ldots e_n) \sqcap ((\mathbf{case}\ e'_0\ \mathbf{of}\ p'_1 : e'_1 | \ldots | p'_k : e'_k)\ e'_{k+1} \ldots e'_n) \\
= ((\mathbf{case}\ e^g_0\ \mathbf{of}\ p_1 : e^g_1 | \ldots | p_k : e^g_k)\ e^g_{k+1} \ldots e^g_n, \bigcup_{i=0}^n \theta_i, \bigcup_{i=0}^n \theta'_i) \\
\quad \text{where} \\
\quad \forall i \in \{0, k+1 \ldots n\}.(e^g_i, \theta_i, \theta'_i) = e_i \sqcap e'_i \\
\quad \forall i \in \{1 \ldots k\}.\exists \theta_i.p_i \equiv (p'_i\ \theta_i) \wedge (e^g_i, \theta_i, \theta'_i) = e_i \sqcap (e'_i\ \theta_i) \\
e \sqcap e' = (x, \{x := e\}, \{x := e'\}) \qquad \text{in all other cases.}
\end{array}
$$

Within these rules, all forms of expression are represented as applications to a set of arguments (possibly the empty set). If both expressions have the same top-level construct, this is made the top-level construct of the resulting generalized expression, and the corresponding sub-expressions within the construct are then generalized. Otherwise, both expressions are replaced by the same fresh variable. It is assumed that the new variables introduced are all different and distinct from the original program variables. The following rewrite rule is exhaustively applied to the triple resulting from generalization to minimize the substitutions by identifying common substitutions that were previously given different names:

$$(e, \theta \cup \{x := e', x' := e'\}, \theta' \cup \{x := e'', x' := e''\})$$
$$\Downarrow$$
$$(e\{x := x'\}, \theta \cup \{x' := e'\}, \theta \cup \{x' := e''\})$$

We define an abstraction operation on expressions that extracts the sub-terms resulting from generalization.

Definition 18 (Abstraction Operation).

$$\begin{array}{l} abstract(e, e') = (\lambda x_1 \ldots x_n.e_0)\ e_1 \ldots e_n \\ \text{where } \ e \sqcap e' = (e_0, \{x_1 := e_1, \ldots, x_n := e_n\}, \theta) \end{array}$$

The correctness of this transformation can be proved locally at each point where generalization takes place by proving the following.

Theorem 4 (Correctness of Generalization).

$$\forall e, e'.abstract(e, e') \overset{\beta *}{\rightsquigarrow} e.$$

Proof. Trivial. Proof of the "abstraction Lemma 2" is immediate from this.

C Residualization

Definition 19 (Residual Program Construction). A residual program can be constructed from a labelled transition system with folding using the rules $\mathcal{C}$ as shown in Figure 8. Here ε is a "function environment", containing the definitions of the functions that appear in the residual program.

$\mathcal{C}[\![e]\!] = \mathcal{C}'[\![e]\!]\ \{\}$

If $e \rightarrow (x, \mathbf{0})(\#1, e_1) \ldots (\#n, e_n)$ then $\mathcal{C}'[\![e]\!]\ \varepsilon = x\ (\mathcal{C}'[\![e_1]\!]\ \varepsilon) \ldots (\mathcal{C}'[\![e_n]\!]\ \varepsilon)$
If $e \rightarrow (c, \mathbf{0})(\#1, e_1) \ldots (\#n, e_n)$ then $\mathcal{C}'[\![e]\!]\ \varepsilon = c\ (\mathcal{C}'[\![e_1]\!]\ \varepsilon) \ldots (\mathcal{C}'[\![e_n]\!]\ \varepsilon)$
If $e \rightarrow (\lambda x, e')$ then $\mathcal{C}'[\![e]\!]\ \varepsilon = \lambda x.(\mathcal{C}'[\![e']\!]\ \varepsilon)$
If $e \rightarrow (\mathbf{case}, e_0)(p_1, e_1) \ldots (p_n, e_n)$ then
$\quad \mathcal{C}'[\![e]\!]\ \varepsilon = \mathbf{case}\ (\mathcal{C}'[\![e_0]\!]\ \varepsilon)\ \mathbf{of}\ p_1 \Rightarrow (\mathcal{C}'[\![e_1]\!]\ \varepsilon) \mid \cdots \mid p_n \Rightarrow (\mathcal{C}'[\![e_n]\!]\ \varepsilon)$
If $e \rightarrow (@, \lambda x_1 \ldots x_n.e_0)(\#1, e_1) \ldots (\#n, e_n)$ then
$\quad \mathcal{C}'[\![e]\!]\ \varepsilon = (\mathcal{C}'[\![e_0]\!]\ \varepsilon)\{x_1 := (\mathcal{C}'[\![e_1]\!]\ \varepsilon), \ldots, x_n := (\mathcal{C}'[\![e_n]\!]\ \varepsilon)\}$
If $e \xrightarrow{\theta} e'$ then $\mathcal{C}'[\![e]\!]\ \varepsilon = (f\ x_1 \ldots x_n)\ \theta$, if $(f\ x_1 \ldots x_n = e') \in \varepsilon$
If $e \xrightarrow{\tau} e'$ then $\mathcal{C}'[\![e]\!]\ \varepsilon = \begin{cases} f\ x_1 \ldots x_n\ \mathbf{where}\ f = \lambda x_1 \ldots x_n.(\mathcal{C}'[\![e'']\!]\ (\varepsilon \cup \{f\ x_1 \ldots x_n = e\})), \\ \qquad \text{if } \exists e''.e'' \xrightarrow{\theta} e \wedge \{x_1 \ldots x_n\} = fv(e) \\ \mathcal{C}'[\![e']\!]\ \varepsilon, \text{otherwise} \end{cases}$

Fig. 8. Rules For Constructing Residual Programs

Within these rules, the parameter ε contains the set of new function calls which have been created, and associates them with the expressions they replace. On encountering a renaming of one of these associated expressions, it is also replaced by an appropriate call of the corresponding function.